UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Diseñador de emails Drag and Drop

Autor/es

RODRIGO TOMÉ NIETO

Director/es

JOSE DIVASÓN MALLAGARAY

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2019-20

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Diseñador de emails Drag and Drop

Realizado por:

Rodrigo Tomé Nieto

Tutelado por:

Jose Divasón Mallagaray

Logroño, Junio, 2020

Índice

1 Resumen

Cada día existen más herramientas para facilitar, mejorar y agilizar el desarrollo, haciéndolo más eficiente. En ciertos casos, cuando una tarea es repetitiva o muy similar pueden desarrollarse soluciones que faciliten el trabajo usando la tecnología drag and drop (arrastrar y soltar).

En el contexto empresarial y del marketing online, los correos electrónicos suelen tener multitud de secciones repetidas y estar todavía basados en una maquetación manual mediante código y tablas. El objetivo del proyecto es crear una herramienta drag and drop que facilite el desarrollo de emails empresariales, aunque pueda aplicarse a otros campos. Por esta razón, hemos desarrollado una herramienta sencilla e intuitiva, para que, a pesar de estar orientada a desarrolladores, personas no técnicas puedan utilizarla.

2 Abstract

Nowadays, there exist many tools which facilitate, improve and speed up software development, making it more efficient. Sometimes, drag and drop technology can be employed to make it easier repetitive and cumbersome tasks

In business and online marketing contexts, emails usually have plenty of repeated sections and are still based on a manual layout using code and tables. The main goal of the project is to create a drag and drop tool that helps with business emails developing, although it can be used in other fields. For this reason, we have developed an easy and intuitive tool, so that, despite being oriented to developers, non-technical people can use it.

3 Introducción

Pixel Division es una empresa dedicada al desarrollo web, plataformas a medida y campañas digitales. Entre su cartera de clientes cuentan con importantes multinacionales, bancos y entidades públicas, como Foster's Hollywood, el Gobierno de La Rioja, ICEMD y BBVA, entre otros.

Estos dos últimos clientes son los más interesantes para el proyecto por el tipo de trabajo que hace para ellos Pixel Division, entre los que está el diseño y desarrollo de correos electrónicos, a lo que a partir de ahora nos referiremos como emailing.

Este trabajo es costoso y complejo debido a que los visores de emails todavía no han adoptado por completo las últimas tecnologías de HTML y CSS.

Por consiguiente, es necesario maquetarlos "a la antigua usanza", es decir, con tablas. Además, los clientes quieren que los correos electrónicos se visualicen correctamente en todos los clientes: móvil, tablet, escritorio e incluso en los navegadores más antiguos como Internet Explorer. Estos requisitos aumentan la complejidad del desarrollo de un correo electrónico, obligando a comprobar cada correo electrónico individualmente en varios visores de emails.

Como cierta mejora del proceso de producción de emailing, Pixel Division cuenta con una serie de snippets para Sublime Text que están comprobados y funcionan correctamente en todos los visores de correo electrónico. A pesar de ello, siguen necesitando comprobar el producto resultante en los distintos visores de emails, utilizando para ello Litmus. Litmus es una aplicación capaz de probar un correo electrónico en una cantidad enorme de clientes de correo electrónico y navegadores.

A pesar de la mejora productiva que ha conseguido el equipo de Pixel Division, el proceso sigue siendo demasiado complejo para que un equipo no especializado pueda desarrollar los correos eléctricos.

3.1 Antecedentes personales

Durante el primer semestre cursé las prácticas obligatorias y optativas en la empresa Pixel Division. Me enseñaron cómo desarrollaban páginas webs, correos electrónicos, temas para Wordpress y React. Elegí esta empresa para realizar las prácticas porque el desarrollo web es el campo al que me gustaría dedicarme, sobre todo la parte de front-end.

En el trascurso de las prácticas asistí a varias charlas que impartieron en la empresa. Una de ellas, en concreto, fue sobre un framework que habían desarrollado para agilizar el proceso de desarrollo web. El framework disponía de una serie de módulos preestablecidos como la cabecera, el pie de página o el destacado.

A raíz de ello se me ocurrió que sería interesante realizar una herramienta que usase el framework para construir páginas webs arrastrando y soltando (drag and drop) los módulos. Se lo propuse y les gustó la idea de poder construir de una manera sencilla los productos que realizan, aunque no les convenció que se usase para desarrollo web, pues sus páginas web suelen tener diseños dispares. Por el contrario, los

correos electrónicos que desarrollan siguen un mismo patrón (dependiente del cliente) por lo que Rubén Lahera, CTO de Pixel Division, me recomendó que la herramienta fuese enfocada para el desarrollo de correos electrónicos.

3.2 Justificación del proyecto

Como hemos dicho, el desarrollo de un correo electrónico todavía es un proceso demasiado complejo para que una "persona no técnica" sea capaz de llevarlo a cabo. Por ello y por otras razones, como puede ser una mayor agilidad en el desarrollo o un menor número de errores en el código, propuse realizar una herramienta con una interfaz drag and drop que facilitase su desarrollo.

Te preguntarás, ¿qué tiene que ver una interfaz drag and drop y en qué ayudará en el problema? La respuesta es sencilla: facilitará todo el desarrollo puesto que contendrá una serie de módulos con los que se puede construir un correo electrónico simplemente arrastrándolos. Una vez situados los módulos, puedes personalizar ciertos aspectos y finalmente exportar el correo electrónico. De esta manera se evita que el programador cometa errores o que un usuario medio sin conocimientos técnicos pueda realizar estos correos.

Objetivo: Realizar una aplicación web con una interfaz de usuario drag and drop a través de la cual se pueda llevar a cabo un correo electrónico, una web o cualquier otro producto basado en HTML, CSS y JavaScript. En concreto, el TFG se va a centrar en el desarrollo de correos electrónicos, pero podría usarse para otros ámbitos dependiendo de los módulos que integre la interfaz.

Como se verá a lo largo de la memoria y especialmente en el capítulo de seguimiento y control, el proyecto sufrió un cambio importante debido a un riesgo no previsto, la pandemia del coronavirus. Esto llevó a la empresa a interrumpir el TFG. Junto con el tutor académico, decidimos seguir con el mismo tema para completarlo sin su supervisión, confiando en que les pudiera resultar útil de todas formas.

4 Planificación

En este apartado vamos a abordar la planificación. Ésta es una de las partes más importantes, puesto que una buena planificación puede ahorrarnos mucho tiempo y dinero en la ejecución del proyecto. Durante esta sección veremos la metodología que usaremos en el proyecto, la estructura de descomposición del trabajo (EDT) y su diccionario, el diagrama de Gantt con las tareas que se van a realizar durante el proyecto y los hitos más importantes y, por último, el plan de gestión de riesgos y comunicación.

4.1 Metodología a seguir

Tras un análisis de las metodologías actuales de realización de proyectos hemos decidido usar la metodología en cascada.

La hemos elegido porque es la que más se ajusta a nuestro proyecto al tener unos requisitos bastante claros. Ésta se compone de varias etapas como puede verse en la ilustración 1. Además, cabe destacar que se podrá repetir el ciclo y que la etapa de verificación se irá realizando poco a poco junto con el cliente.

1. **Análisis**: en esta etapa se capturan los requisitos del cliente y el alcance del proyecto. Además, de definir el problema a tratar.
2. **Diseño**: se toman las decisiones sobre la arquitectura de la aplicación, el diseño de la interfaz y otras decisiones necesarias para llevar a buen puerto la herramienta.
3. **Implementación**: durante esta etapa se llevará a cabo el desarrollo de la aplicación siguiendo las directrices que se han marcado en la etapa de diseño.
4. **Verificación**: por último, se comprueba el correcto funcionamiento del sistema y la conformidad del cliente con el producto obtenido.

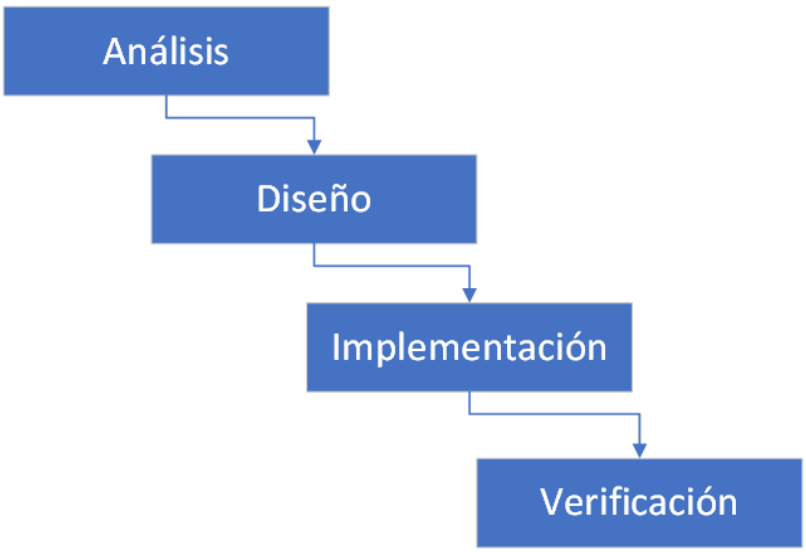

Ilustración 1: Esquema de metodología en cascada

4.2 EDT

La EDT se compone de ocho bloques de trabajo que a su vez se dividen en tareas más pequeñas si es preciso, tal y como se ve en la ilustración 2.

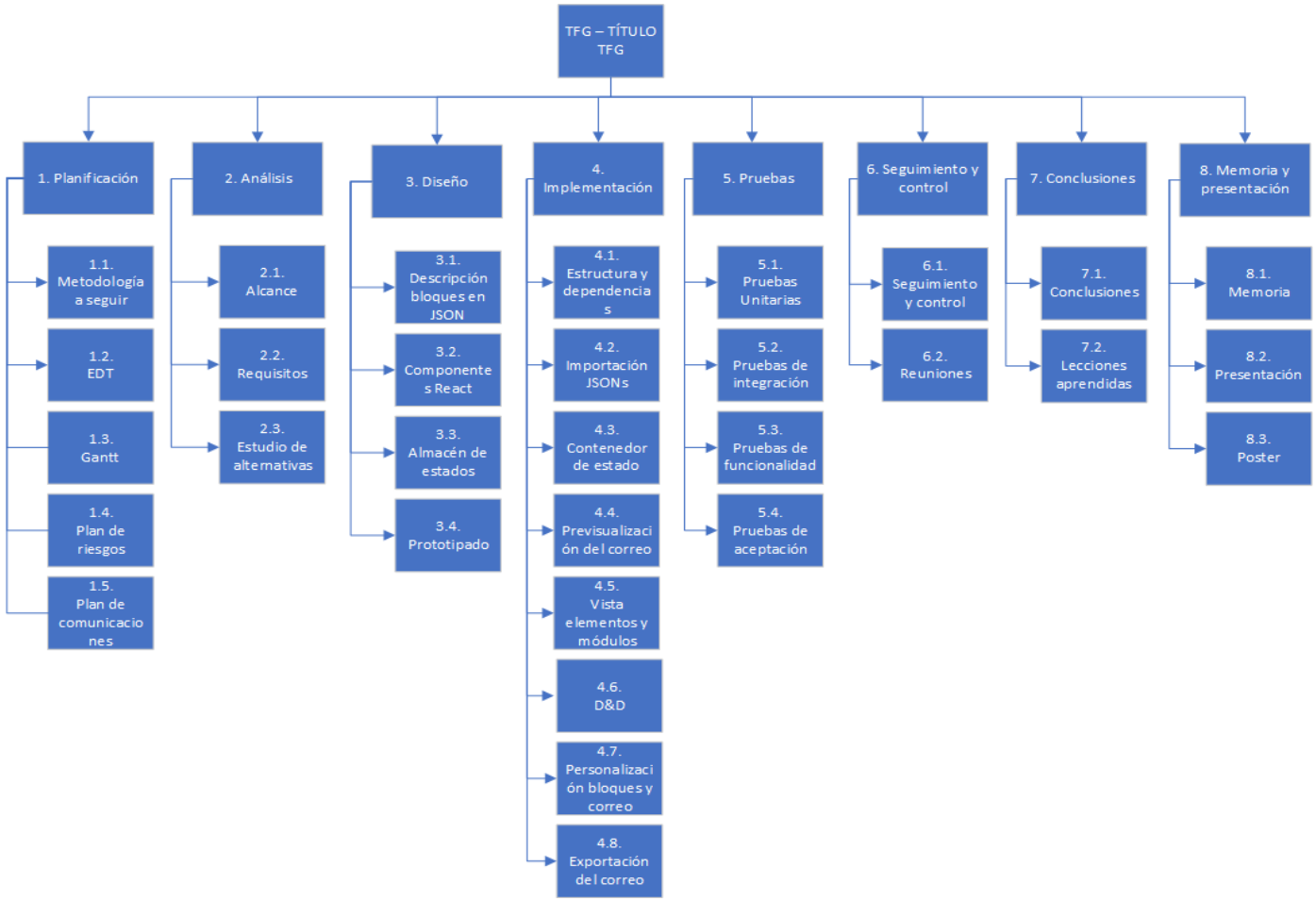

Ilustración 2: EDT del proyecto

4.3 Diccionario de la EDT

El diccionario de la EDT consiste en la explicación de los paquetes de trabajo de los que se compone la EDT y de sus tareas. Puede verse la descripción de cada uno de los componentes de la EDT en la siguiente tabla.

ID	Nombre	Descripción
1.	Planificación	
1.1.	Metodología a seguir	Elección y explicación de la metodología que se va a seguir
1.2.	EDT	Estructura de descomposición del trabajo del proyecto y su correspondiente diccionario
1.3.	Gantt	Planificación y distribución de las tareas del proyecto, además de la estimación y duración de estas
1.4.	Plan de riesgos	Descripción del plan de riesgo que se va a seguir en el proyecto
1.5.	Plan de comunicaciones	Descripción del plan de comunicaciones del proyecto
2.	Análisis	
2.1.	Alcance	Definición de la envergadura del proyecto
2.2.	Requisitos	Descripción de los requisitos funcionales y no funcionales
2.3.	Estudio de alternativas	Breve descripción de las principales alternativas y los motivos por las que se aceptan/rechazan
3.	Diseño	
3.1.	Descripción bloques en JSON	Diseño del modo de representación de los bloques en JSON
3.2.	Componentes React	Definición de los componentes y contenedores necesarios
3.3.	Almacén de estados	Definición del estado a guardar y operaciones necesarias
3.4.	Prototipado	Diseño preliminar de la interfaz de usuario del proyecto
4.	Implementación	
4.1.	Estructura y dependencias	Organización del proyecto y dependencias conocidas
4.2.	Importación JSONs	Descripción del método que se ha usado para importar los bloques
4.3.	Contenedor de estado	Explicación de la implementación del contenedor de estado
4.4.	Previsualización del correo	Descripción de la técnica usada para previsualizar el correo
4.5.	Vista elementos y módulos	Desarrollo componente de visualización de bloques
4.6.	D&D	Implementación de la funcionalidad D&D
4.7.	Personalización bloques y correo	Descripción de los componentes necesarios, opciones que se permiten personalizar e implementación
4.8.	Exportación del correo	Descripción del método usado para exportar el correo en HTML
5.	Pruebas	
5.1.	Pruebas unitarias	Descripción de las pruebas unitarias que se realizarán
5.2.	Pruebas de integración	Descripción de las pruebas de integración que se realizarán
5.3.	Pruebas de funcionalidad	Descripción de las pruebas de funcionalidad que se realizarán
5.4.	Pruebas de aceptación	Descripción de las pruebas de aceptación que se realizarán
6.	Seguimiento y control	
6.1.	Seguimiento y control	Seguimiento y control del cumplimiento de la planificación
6.2.	Reuniones	Reuniones con el interesado y el tutor
7.	Conclusiones	
7.1.	Conclusiones	Extracción de las conclusiones más importantes del proyecto
7.2.	Lecciones aprendidas	Determinar y analizar los elementos que han tenido éxito o no.
8.	Memoria y presentación	
8.1.	Memoria	Realización de la memoria del proyecto
8.2.	Presentación	Realización de la presentación y preparación de la defensa
8.3.	Poster	Realización del poster del proyecto

4.4 Diagrama de Gantt

En el diagrama de Gantt (ilustración 3) se puede ver la distribución de la carga de trabajo a lo largo de las dieciocho semanas del proyecto en base a las horas estimadas necesarias para completar cada paquete de la EDT. Además, pueden verse los hitos más importantes del proyecto.

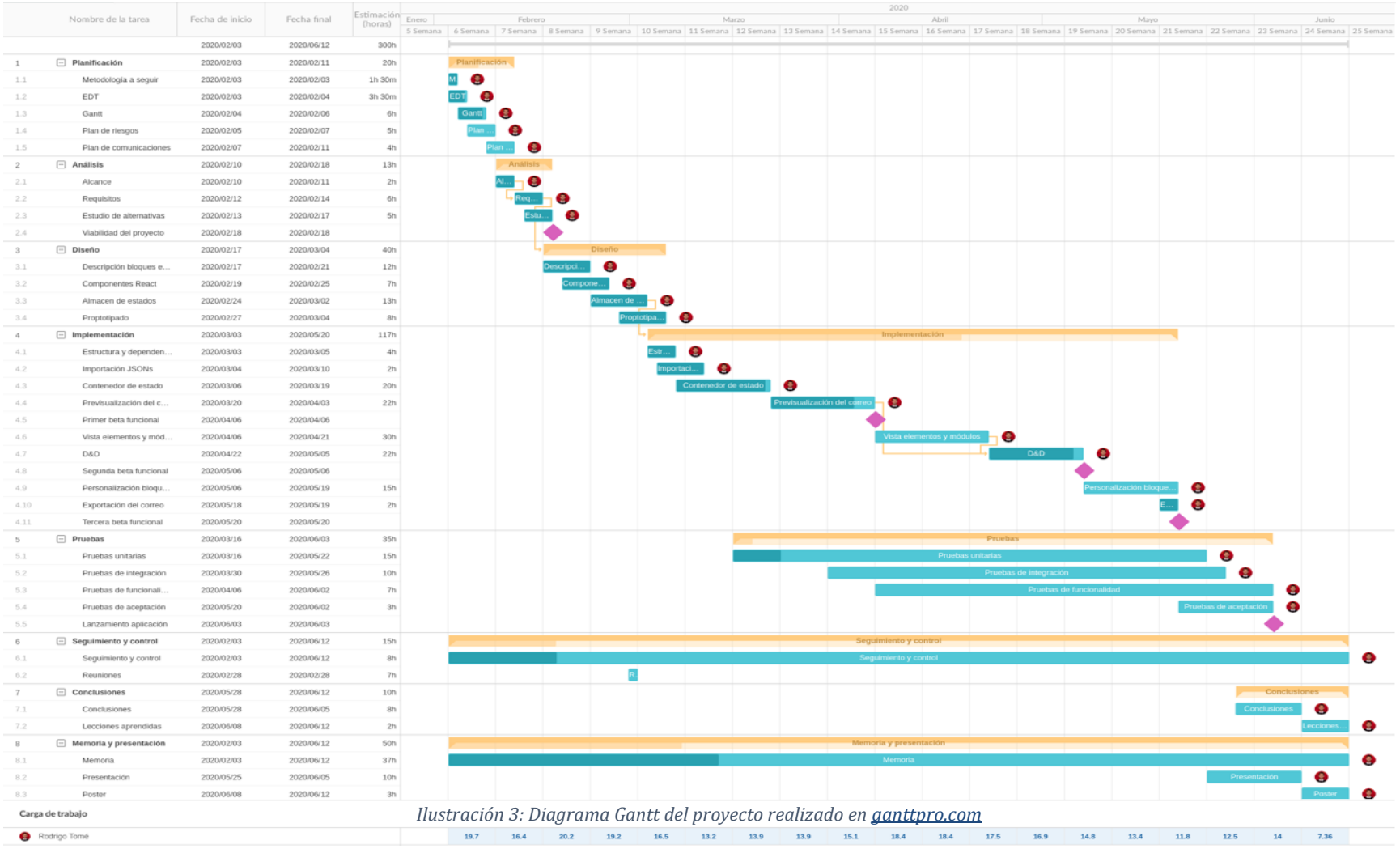

Ilustración 3: Diagrama Gantt del proyecto realizado en ganttpro.com

4.5 Plan de riesgos

Antes de comenzar con el plan de riesgos es necesario conocer a las partes implicadas en este proyecto:

- **Destinatario**: El destinatario es la empresa Pixel Division. Más concretamente, el representante de la empresa Rubén Lahera.
- **Tutor académico**: El tutor de este proyecto es Jose Divasón, profesor del área de Ciencia de la Computación e Inteligencia Artificial de la Universidad de La Rioja.

Para detectar los posibles riesgos del proyecto, primeramente, hemos realizado una búsqueda bibliográfica que nos ha permitido identificar una serie de riesgos comunes en la gestión y desarrollo de cualquier proyecto informático (Ramón Rodríguez (coordinador), García Mínguez, & Lamarca Orozco, 2007). Para reducirlos seguiremos las correspondientes estrategias de prevención, minimización y contingencia. Además, debido a que este proyecto es un TFG académico, tiene una serie de riesgos más específicos que no están incluidos en la cita anterior. Los hemos identificado y son presentados en la siguiente tabla.

Riesgo	Estrategia de prevención	Estrategia de minimización	Plan de contingencia
Insatisfacción con el proyecto	Comunicación continua con el cliente para comprender lo que necesita	Comunicación continua con el cliente	No es posible al haberse consumido todas las horas
Ausencia del tutor	No puede preverse	Comunicación continua con el cliente	Establecer comunicación a través de medios telemáticos
Pérdida de documentación o código	Uso de herramientas que permitan la sincronización con la nube	No es posible minimizarlo	Recuperar la documentación o el código de la nube
Enfermedad	Uso de herramientas que permitan la sincronización con la nube	Comunicación continua con el cliente y tutor por medios telemáticos	Replanificar

4.6 Plan de comunicaciones

Al igual que en el plan de riesgos, en el plan de comunicaciones existen diferentes actores, pero en nuestro caso son los mismos que en el plan de riesgo, por ello no los volveremos a citar. La comunicación puede ser síncrona o asíncrona y generalmente se llevará a cabo por uno de los siguientes motivos: informar, solicitar información o llegar a un acuerdo.

La comunicación con el tutor de empresa será principalmente síncrona y con el académico asíncrona, pero cada tres semanas existirá una reunión síncrona. Puede verse a continuación con más detalle:

Tipo de comunicación	Informar	Solicitar información	Llegar a un acuerdo
Síncrona	Reunión semanal	Reunión semanal	Reunión semanal
Asíncrona	Correo electrónico	Correo electrónico	Correo electrónico

5 Análisis

5.1 Alcance

Este proyecto tiene como principal objetivo construir una interfaz de usuario con la que poder desarrollar correos electrónicos con solo arrastrar bloques. De esta manera se conseguirá mejorar el proceso productivo de la empresa Pixel Division, a la vez que se posibilita que alguien no técnico pueda realizar correos electrónicos.

5.2 Enfocando nuestro TFG

En lo que respecta a nuestro TFG, debemos recordar que el principal objetivo de la herramienta es la construcción de correos electrónicos. Es por ello por lo que es necesario definir y dar forma a los "bloques" que usará la herramienta. En concreto es posible diferenciar dos tipos de bloques.

El primero, los elementos. Los elementos son la unidad básica de construcción, es decir, sin estos no se podrían formar estructuras más complejas. Entre los cuales se encuentran imagen, call to action (botón o enlace), texto, línea de separación y espaciador.

Como segundo tipo de bloque tenemos los módulos. Estos son estructuras complejas que se repiten con frecuencia en los correos electrónicos. Están formados por elementos. En esta categoría podemos encontrar la cabecera, pie de correo electrónico, bloques de estructuración de contenido (dos columnas, tres columnas, etc.) …

Una vez vistos los tipos de bloques y entendiendo que la mayor parte de los módulos pueden ser construidos con conjuntos de elementos, debemos destacar que no todos los módulos entran en el alcance de este proyecto. Solo se incluirán el módulo contenedor, cabecera, pie y bloques de estructuración de contenido.

Cabe destacar que la tecnología principal a usar será React. Propusimos alternativas como Angular o Vue, pero el cliente escogió React, una biblioteca JavaScript para construir interfaces de usuario, al tener más conocimiento en esta tecnología y ser de su preferencia. Además de React se usarán tecnologías ya establecidas como HTML, CSS y JavaScript.

5.3 Requisitos

A continuación, se recogen los distintos requisitos del proyecto (acordados tras varias reuniones con el cliente), tanto funcionales como no funcionales. Además de posibles ampliaciones.

- Requisitos funcionales:
    - Crear una herramienta con la cual poder desarrollar correos electrónicos
    - Los correos electrónicos deben poder exportarse en HTML

    - Los elementos deben poder personalizarse:
        - Texto: color de fuente y tamaño de la letra
        - Call to action (CTA): color de fuente, color de fondo, tamaño de la letra y URL
        - Imagen: URL y descripción del atributo HTML ALT
        - Línea de separación: color línea y grosor de la línea
        - Espaciador: color de fondo y grosor del espaciador
    - Los bloques deben ser contenidos en la aplicación
    - Debe existir una lista con los bloques disponibles y un espacio donde poder desarrollar el correo electrónico
    - El correo electrónico debe tener una configuración donde establecer un texto de previsualización y el color de fondo
- Requisitos no funcionales:
    - La aplicación debe estar desarrollada en React
    - Los bloques deben ser almacenados en JSON
    - El correo electrónico generado por la herramienta debe verse correctamente en todos los clientes principales (en caso de que se usen los bloques de Pixel Division)

A estos requisitos vistos anteriormente proponemos añadir los siguientes, además de unas posibles ampliaciones:

- Requisitos no funcionales adicionales:
    - La interfaz se compone de dos columnas. En una de ellas se encontrará el listado de módulos y en la otra se podrán arrastrar los bloques para desarrollar el email
    - Se abstraerá la herramienta para que no dependa de los bloques
- Posibles ampliaciones:
    - Soporte de nuevos bloques para correos electrónicos
    - Exportar/Importar los correos electrónicos en JSON
    - Compatibilidad con plantillas de correos electrónicos
    - Guardar los diseños realizados
    - Recogida de los bloques vía API
    - Verificación de los bloques mediante JSON schema
    - Compatibilidad con otro tipo de bloques para aumentar la funcionalidad y poder usarse para desarrollar páginas webs o cualquier otro producto basado en HTML, CSS y JavaScript

5.4 Estudio de alternativas

Con el objetivo de decidir la viabilidad del proyecto se realizó una búsqueda de aplicaciones existentes que cumpliesen el alcance y requisitos definidos. Se encontraron varias alternativas, aunque ninguna cumplía con las exigencias del proyecto:

- Cumplir el alcance del proyecto, estar actualizada y tener soporte
- Posibilidad de incluir módulos propios
- Soporte para todos los clientes de correo electrónico
- Gratuidad de la herramienta

Las alternativas encontradas fueron las siguientes:

1. ContentBuilder.js: ContentBuilder.js es el primer editor que usa drag and drop y permite construir el diseño de su producto en casi cualquier framework CSS como en Bootstrap, Foundation o Materialize.

   Exigencias del proyecto que incumple:

   a. Está orientado a web no a correos electrónicos
   b. No se pueden incluir tus propios módulos
   c. No es gratuito

2. Builder.io: Editor visual integrable en sitios webs y aplicaciones. Dispone de una API REST con la que hay que conectarse para que funcione el editor. Además, soporta varios frameworks como React, Angular… Dispone de bloques especiales para correos electrónicos, pero estos están en construcción.

   Exigencias del proyecto que incumple:

   a. Está orientado a web no a correos electrónicos
   b. No se pueden incluir tus propios módulos
   c. No es gratuito

3. Structor: Herramienta de prototipado de interfaces capaz de generar el código fuente de la interfaz en React. Permite crear tus propios bloques y dispone de un Marketplace de plugins para aumentar su funcionalidad.

   Exigencias del proyecto que incumple:

   a. Esta obsoleta y sin soporte

Tras analizar las alternativas encontradas se ha decidido llevar a cabo el proyecto debido a que las herramientas que se han descrito carecen de la posibilidad de crear tus propios bloques, son de pago y no están orientadas a correos electrónicos.

Como excepción, encontramos Structor que, aunque sí soporta la creación de bloques propios y es gratuita, es demasiado compleja, se encuentra obsoleta y lleva dos años sin soporte.

6 Diseño

En esta sección vamos a ver el diseño de los distintos apartados que componen la aplicación. En concreto, diseñaremos el modelo de importación de bloques, los componentes y contenedores, el almacén de estados y por último el prototipo de la interfaz de usuario.

6.1 Importación de bloques

Es necesario estudiar y diseñar cómo será el modelo de importación de bloques en la herramienta.

Por requisito del cliente los bloques deben ser descritos en JSON. Cada bloque debe describir un fragmento de HTML perteneciente a un elemento o un componente.

Tras analizar el problema hemos llegado a dos posibles soluciones:

- JSON describe las componentes básicas del bloque (nombre, icono…) y guarda el HTML en base64

```
{
    "name": "Line",
    "icon": "icon-line",
    "category": "element",
    "custom": {
        "bgcolor": "#F4F4F4"
    },
    "children": [[]],
    "content": "PHRyPjx0ZCBhbGlnbj0i"
}
```

- JSON describe las componentes básicas del bloque (nombre, icono…) y describe el HTML usando JSON

```
{
    "name": "Line",
    "icon": "icon-line",
    "category": "element",
    "custom": {
        "bgcolor": "#F4F4F4"
    },
    "children": [[]],
    "content": {
        "elementType": "tr",
        "content": [
            {
                "elementType": "td",
                "elementConfig": {
                    "align": "center"
                    "style": {
                        "font-size": 0
                    }
                }
            }
        ]
    }
}
```

Después de evaluar las dos posibles opciones nos hemos decantado por la primera.

Las razones son que, tras unas pequeñas pruebas, el costo de transformar la descripción del HTML en JSON a JSX (una extensión de la sintaxis de JavaScript recomendada para React) es mayor a descodificar el HTML de base64. Aparte de esto, esta opción facilita adaptar los bloques existentes a este formato.

A pesar de estas ventajas cabe destacar que existe una desventaja importante, y es que si alguien nuevo se incorpora al proyecto no sabrá qué código HTML contiene cada bloque. Para minimizar el impacto de este problema realizamos una documentación en la que se incluye el código de cada bloque.

6.2 Definición de componentes y contenedores

Una vez solventado el problema de cómo importar los bloques a la herramienta, es importante definir qué componentes (permiten separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada) y contenedores (componentes con lógica) serán necesarios para llevar a cabo el proyecto. De esta manera nos resultará más sencillo el proceso de implementación.

Contenedores:

- **Aplicación**: contenedor principal de la aplicación. En él se definirán las rutas de la aplicación
- **Diseño**: contendrá todo lo relacionado con la vista de diseño. Se encargará de contener la Previsualización y la Barra lateral
- **Previsualización**: aquí se arrastrarán los bloques que se encontrarán en la Barra lateral. Una vez arrastrado se visualizará el bloque
- **Barra lateral**: servirá de contenedor para las distintas pestañas de la aplicación

Componentes:

- **Bloque visual**: es el componente que contendrá la parte visible de los bloques, será parte de las pestañas de orden
- **Pestaña**:
    - **Orden**: permitirá la ordenación de los bloques según su tipo (elemento o componente)
    - **Personalización**: permitirá personalizar un bloque

6.3 Almacén de estados inmutables

Después de discernir los componentes y contenedores de la herramienta es necesario definir cómo vamos a almacenar el estado de la aplicación.

El estado de la aplicación describe cómo se encuentra la aplicación en cada momento. En nuestro caso, grosso modo, podríamos decir que la aplicación viene descrita por los bloques, el contenedor principal y el correo electrónico construido.

La información almacenada en el estado debe estar disponible a nivel de aplicación para, de esta manera, acceder a los datos desde cualquier componente o contenedor. Para resolver este problema existen dos posibles soluciones: una opción es usar la API de contexto nativa de React u otra opción, como Redux (una librería JavaScript de código abierto para el manejo del estado de una aplicación), de la cual hablaremos más adelante.

La API de contexto de React no requiere de librerías externas, al contrario que Redux que sí que requiere de dos paquetes `npm` independientes, `redux` y `react-redux`. Además, cabe destacar que el uso de Redux es un poco más complejo que la API nativa.

También nos gustaría mencionar que Redux se comporta mejor cuando se modifica el estado frecuentemente ya que no necesita renderizar toda la aplicación.

Por este motivo hemos decido usar Redux para la realización de este proyecto en vez de la API nativa de contexto.

Sabiendo ya el porqué es necesario un almacén de estados y habiendo elegido qué implementación vamos a usar, es necesario decidir qué se va a guardar en el almacén. En concreto, almacenaremos la siguiente información:

- Los módulos de la herramienta
- Los elementos de la herramienta
- Los bloques que se han dispuesto en la previsualización
- El bloque activo (por ejemplo, al clicar en uno para editar sus propiedades)
- El contenedor base (en nuestro caso el de un email)

6.4 Prototipado

El prototipado de la interfaz ha sido relativamente sencillo puesto que se ha basado en un diseño propuesto por los interesados, quienes destacaban de este diseño su sencillez y ser intuitivo. Puede verse en la ilustración 4.

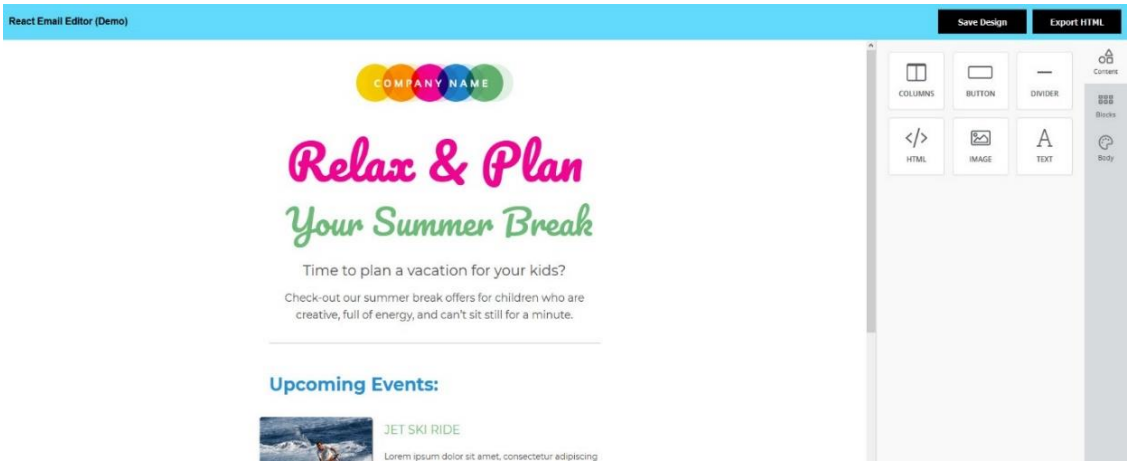

Ilustración 4: React Email Editor

Recordemos que se estipulaba como requisito que existiera un listado de los bloques y un entorno donde poder realizar el diseño del correo electrónico. Por esta razón se ha optado por un diseño en dos columnas: una para diseñar el correo y otra con el listado de bloques.

Respecto al listado de bloques se organizará teniendo en cuenta el tipo de bloque (elemento o módulo). Además, contendrá otra sección donde estarán las opciones de personalización del correo.
La parte más complicada de esta sección ha sido la elección de la iconografía que representa los bloques.

En concreto, este proyecto no dispondrá de una interfaz móvil al ser la herramienta demasiado compleja como para contenerla en un espacio tan reducido.

El diseño completo de la interfaz puede verse en las ilustraciones 5, 6 y 7.

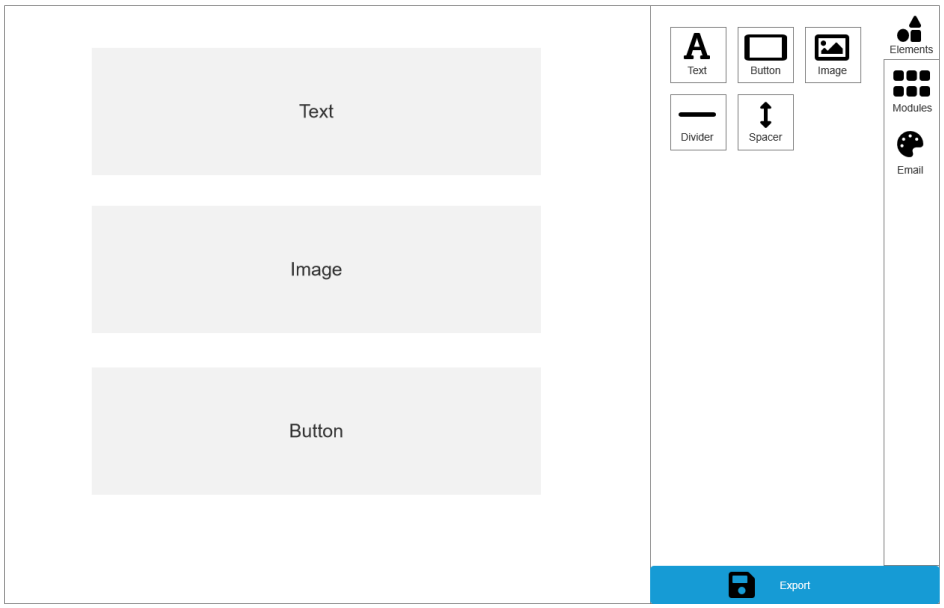

Ilustración 5: Página principal

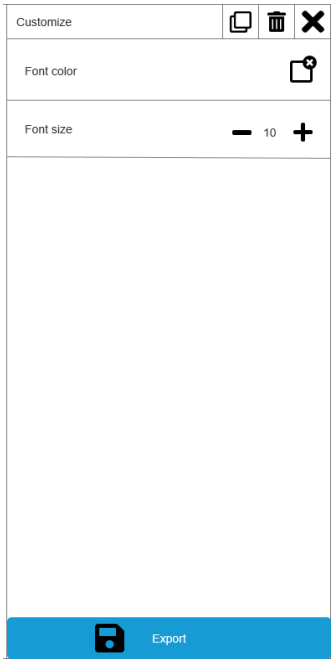

Ilustración 6: Personalización del correo electrónico

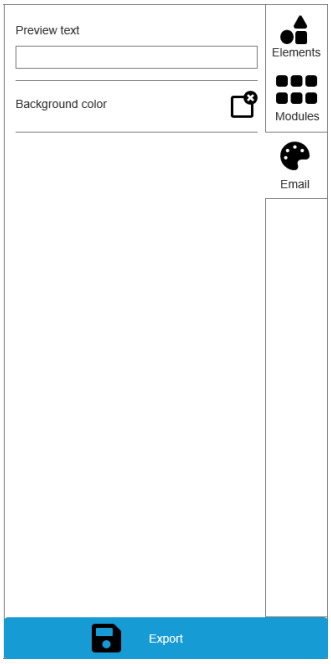

Ilustración 7: Personalización de un bloque

7 Implementación

En esta sección vamos a ver el proceso de implementación que se ha llevado a cabo. Este proceso está dividido en cuatro fases: almacén de estados, componentes y funcionalidad, método de renderizado del JSON y desarrollo de la interfaz.

Dentro de cada apartado se describirá qué se ha hecho, cómo se ha realizado el proceso de implementación y los problemas que han surgido en caso de que hayan surgido alguno. De ser así también se describirán las alternativas consideradas y la solución final adoptada.

7.1 Tecnologías

El desarrollo de esta herramienta se ha realizado en Windows, debido a que como desarrolladores nos encontrábamos más cómodos en esta plataforma y no aporta ventajas suficientes el uso de Linux como para realizar el cambio.

Para llevar a cabo el proyecto se han usado las siguientes tecnologías:

- NPM como gestor de paquetes, puesto que permite una fácil instalación y gestión de dependencias
- `create-react-app` paquete NPM de Facebook para la puesta en marcha rápida de un proyecto basado en React. Incluye Babel como compilador (ej.: ES6) y WebPack como gestor de tareas (ej..: preprocesado)
- Visual Studio Code como editor de texto
- SASS como extensión de CSS
- Standard como guía de estilo para JavaScript y SMACSS como orden de propiedades CSS
- FontAwesome como paquete NPM para la inclusión de iconos
- GitHub Actions para realizar integración continua, cuando se realiza un push sobre la rama master del repositorio se lanza la acción e instala Node, las dependencias y pasa los test, además puede añadirse el despliegue automático en caso de que sea necesario
- Redux como almacén de estados
- `react-color`, paquete NPM que ofrece un componente en React para seleccionar un color
- React Developer Tools, es una extensión para el navegador que permite ver los componentes de React y sus atributos
- `redux-thunk`, paquete NPM que nos permite aplicar un middleware a Redux
- `css-declaration-sorter`, paquete NPM que nos permite ordenar automáticamente las propiedades CSS

7.2 Almacén de estado (Redux)

Esta fase consiste en implementar el "lugar" donde se guardarán los estados por los que va pasando la aplicación. En concreto, se implementará el almacén usando la librería Redux, como ya elegimos en la etapa de diseño.

Antes de comenzar con la explicación de cómo se ha llevado a cabo la implementación del almacén queremos aclarar que la estructura de carpetas elegida para este propósito puede verse en la siguiente ilustración.

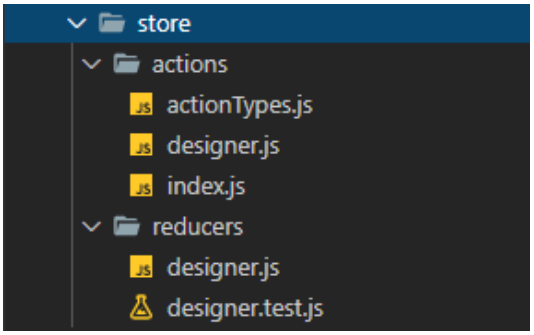

Ilustración 8: Estructura de carpetas del almacén de estados

7.2.1 Implementación

React por lo general usa estados inmutables, de lo contario los estados pueden mutar de manera impredecible. El estado puede existir a nivel de componente o a nivel de aplicación. Compartir el estado entre componentes es fácil mientras tengan una relación padre-hijo, en caso contrario la cosa se complica. La siguiente ilustración muestra el problema (ilustración 9):

En el lado izquierdo, se puede ver un árbol de componentes de React. Una vez que un componente inicia el cambio de estado, el nuevo estado necesita ser propagado a los demás componentes que dependen de los datos modificados.

En esta situación es donde Redux es útil. Redux es una librería JavaScript para gestionar el estado de una aplicación de una manera predecible y sencilla. El estado se encuentra en un almacén y los componentes *"vigilan"* los datos que les interesan del almacén.

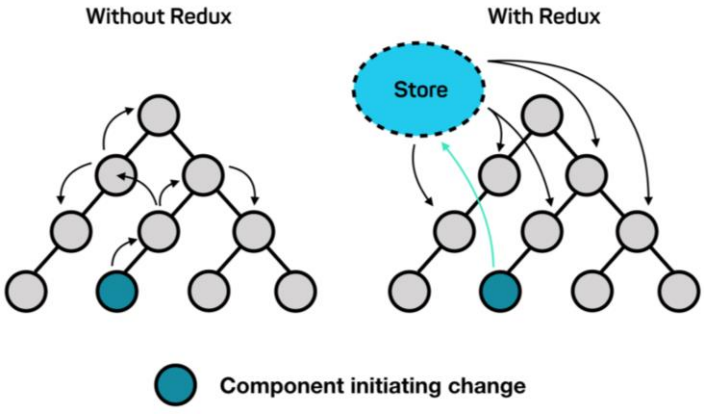

Ilustración 9: Problemática al compartir un estado

7.2.1.1 Patrón Flux

El patrón Flux puede implementarse de distintas maneras, el patrón original o arquitectura fue diseñado por Facebook para ayudarles a manejar todo el flujo de trabajo de React.

Existen dos principales diferencias entre la librería Flux y Redux. Flux permite tener varios almacenes y mutar los estados. En cambio, Redux solo maneja un almacén centralizado que, aunque es posible crear más de uno, al final deben combinarse para formar un solo almacén, y no es posible mutar los estados.

La inmutabilidad y predictibilidad del almacén se consigue gracias a que los componentes no interactúan directamente con el almacén, sino que, para modificarlo le envían acciones. Estas pueden considerarse como la "intención de hacer algo" puesto que hasta que no llegan a los reductores no se produce la operación. Con esto conseguimos que las operaciones sean predecibles y se puedan depurar fácilmente.

Por último, para que todo funcione los componentes deben suscribirse (patrón Observer) a las partes del estado que necesitan y "reaccionan" en consecuencia con los cambios de estado.

En resumidas cuentas, las ventajas que nos aporta este flujo de trabajo son: una forma sencilla de gestionar el estado a nivel de aplicación puesto que es posible transmitir el estado a cualquier componente, aunque no exista una relación padre-hijo, un estado predecible y una manera sencilla de depurar la aplicación.

Puede ver todo este flujo de trabajo a continuación (ilustración 10).

Posteriormente realizaremos una explicación más detallada de las distintas partes con unos pequeños fragmentos de código.

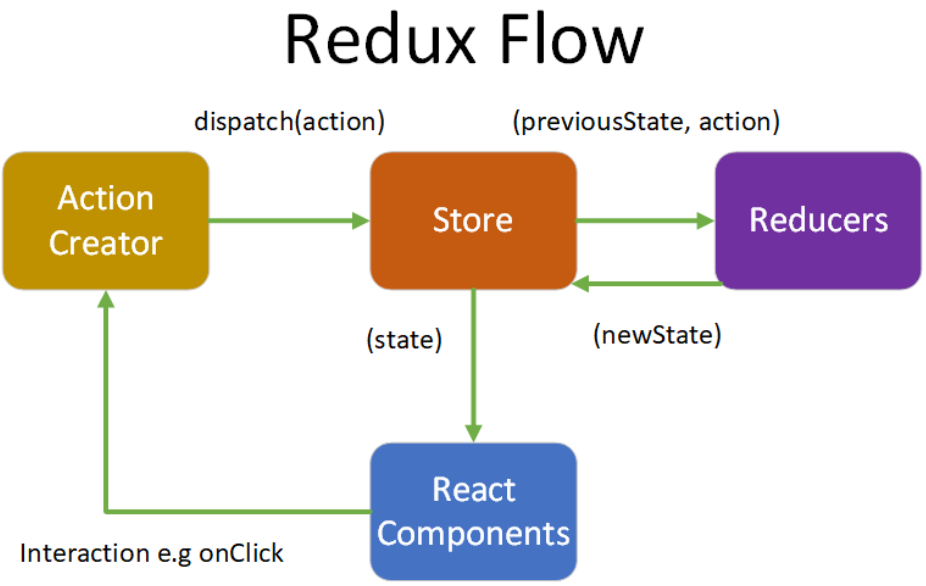

Ilustración 10: Patrón Flux

7.2.1.2 Estado

El estado de Redux contiene todos los datos de la aplicación. Este estado es básicamente un árbol de objetos (patrón Singleton) al que se puede acceder desde todos los componentes de la aplicación. En concreto, nuestra aplicación contiene un pequeño estado inicial que se encuentra en **store/reducers/designer.js**. En este estado se guardan todos los datos importantes, que se seleccionaron en la etapa de diseño, y que necesitan ser compartidos entre los componentes, como ya hemos mencionado antes.

```
const initialState = {
  modules: [],
  elements: [],
  previewBlocks: [],
  blockActive: null,
  baseContainer: null
}
```

7.2.1.3 Acciones

Las acciones son objetos JavaScript simples, consisten en una propiedad `type` obligatoria para poder identificar la acción y la información adicional que necesitemos. El tipo debe de ser un `string`, normalmente están almacenadas en constantes para no equivocarse al escribirlas y así tenerlas todas juntas en un fichero separado aumentando la claridad. Por otro lado, no existe una especificación para la implementación de los objetos con información adicional, nosotros en concreto lo hemos hecho así:

Los tipos de acciones se encuentran en: **store/actions/actionTypes.js**

```
export const INIT_ELEMENTS = 'INIT_ELEMENTS'
```

Por otro lado, las acciones propiamente dichas, se encuentran en: store/actions/designer.js

```
export const initElements = (elements) => {
  return {
    type: actionTypes.INIT_ELEMENTS,
    elements: elements
  }
}
```

Más concretamente, el código que se ha mostrado no es exactamente una acción si no un *"creador de acción"*, más comúnmente conocido como `action creator`. El uso de los creadores de acciones en vez de llamar directamente las acciones nos ofrece mayor flexibilidad y nos facilita realizar pruebas posteriormente.

Lo que hemos visto hasta ahora permite hacer creadores de acciones sencillos, pero si queremos algo más de flexibilidad deberemos de usar el método `dispatch` que nos permite lanzar acciones pasándoselas como parámetros, esto nos permitirá lanzar uno u otro creador de acciones.

Si además queremos lanzar varios creadores de acciones, como ocurre en nuestro caso, que queremos lanzar con una sola función `loadBlocks`, los creadores de acciones `initElements` y `initModules` es necesario ir un paso más allá e incorporar un middleware para poder efectuar operaciones asíncronas. En concreto hemos incluido el middleware `redux-thunk` lo que nos

permite tener acciones asíncronas o despachar por ejemplo dos creadores de acciones (el código ha sido simplificado).

```
export const loadBlocks = () => {
  const elements = importAll('ruta')
  const modules = importAll('ruta')

  return dispatch => {
    dispatch(initElements(elements))
    dispatch(initModules(modules))
  }
}
```

7.2.1.4 Reductor

Los reductores son sencillamente una función que, dado un estado previo (en caso de que no exista usan un estado inicial) y la acción a realizar devuelven el siguiente estado.

Lo más importante de los reductores es que no deben modificar el estado actual, puesto que es inmutable, por ello es necesario realizar una copia del estado y posteriormente modificarlo. En caso de que no se realice así el cambio de estado, podemos encontrarnos con una aplicación inestable e impredecible.

En nuestro caso, decidimos realizar la copia del estado actual mediante el uso del objeto JavaScript JSON, convirtiendo el estado a formato JSON y posteriormente volviendo a construir el objeto. De esta manera se consigue una copia profunda del objeto de una forma sencilla.

Nuestro reductor se encuentra en: **store/reducers/designer.js**

```
const reducer = (state = initialState, action) => {
  let updatedState = deepcopy(state)

  switch (action.type) {
    case actionTypes.INIT_ELEMENTS:
      updatedState = initElements(state, action, updatedState)
    default:
      console.log('[Designer Reducer] Action default case')
      break
  }

  return updatedState
}
export default reducer
```

El código anterior ha sido simplificado, pero es suficiente para entender la idea. Además, para simplificar el reductor se han extraído las funciones que aplican las acciones.

Para acabar con los reductores es posible, incluso recomendable, separar los reductores en distintos ficheros y combinarlos con la función `combineReducers`, para que posteriormente sea posible crear el almacén.

7.2.1.5 Almacén

El almacén contiene el árbol de estado de la aplicación. La única forma de cambiar el estado que contiene es llamando a una acción.

Nuestro almacén se inicializa fuera la estructura de carpetas vista anteriormente, en concreto se realiza en el fichero **index.js** que se encuentra en la raíz de la carpeta **src**.

```
const store = createStore(reducer)
```

7.2.1.6 Redux con React

React puede acceder al almacén de dos formas, la primera es usando el método `store.subscribe()` o usando la función `connect()()`. Hemos elegido la segunda forma porque tiene la ventaja de que previene renderizados innecesarios.

El método connect se diseñó siguiendo el patrón `presentational/container` lo que significa que el componente que realizamos nosotros es la vista. Posteriormente cuando usamos el método connect creamos un componente contenedor.

A la función connect se le pasa en el primer paréntesis la parte del estado que necesita el componente (`mapStateToProps`) y los creadores de acciones que use (`mapDispatchToProps`). En el segundo paréntesis debemos pasarle el componente al que queremos que añada `mapStateToProps` y `mapDispatchToProps` como parámetros resultando la llamada algo tal que así:

```
connect(mapStateToProps, mapDispatchToProps)(App)
```

7.2.1.7 Debug Redux

Redux nos da un estado transparente y predecible, la única manera de cambiarlo es llamando un creador de acciones. Gracias a esto podemos crear un middleware y añadirlo a la función que crea el almacén para poder observar los cambios de estado. Además, de añadir este middleware necesitamos aplicar el middleware `redux-thunk` para permitir las operaciones asíncronas en las acciones como ya dijimos en esa sección.

Para facilitar aún más la depuración de Redux vamos a usar la extensión para navegadores Redux Dev Tools que nos permite comprobar de manera sencilla el estado actual, los anteriores e incluso hacer *"retroceder en el tiempo"* a la aplicación. Redux Dev Tools necesita modificar la función `compose` que sirve para combinar funciones.

El resultado final del código (ha sido simplificado) es el siguiente:

```
const composeEnhancers = REDUX_DEVTOOLS_EXTENSION_COMPOSE || compose
const logger = store => {
  return next => {
    return action => {
      console.log('[Middleware] Dispatching', action)
      const result = next(action)
      console.log('[Middleware] Next State', store.getState())
      return result
    }
  }
}
const store = createStore(reducer, composeEnhancers(applyMiddleware(t
hunk, logger)))
```

Gracias a esto conseguimos que en la consola salgan los estados por los que ha pasado la aplicación y las acciones que se han producido, además de que nos funcione el plugin Redux Dev Tools.

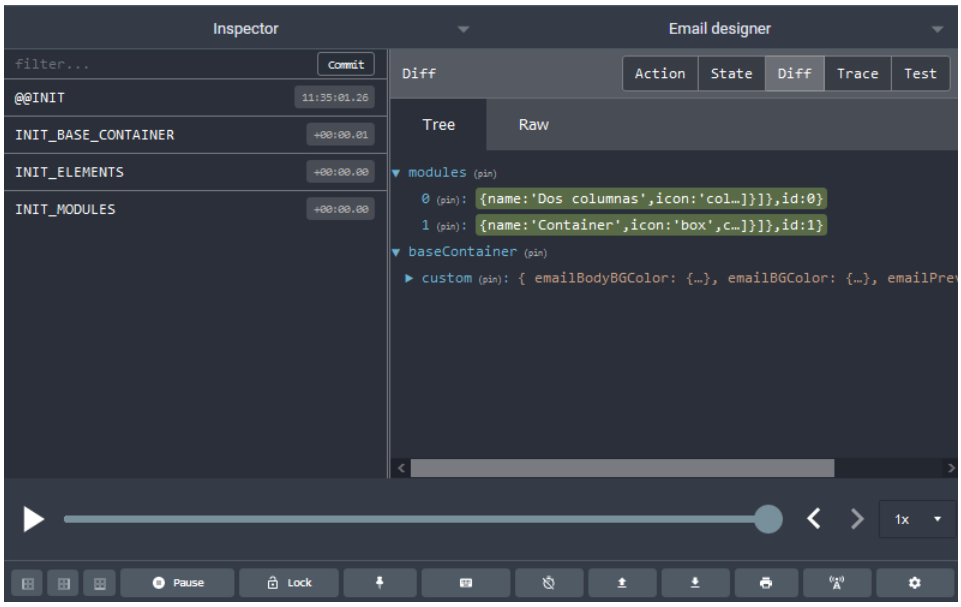

Ilustración 11: Redux Dev Tools

```
[HMR] Waiting for update signal from WDS...
[Designer Reducer] Action default case
[Middleware] Dispatching ▶ Object { type: "INIT_BASE_CONTAINER", container: {…} }
[Middleware] Next State
▶ Object { modules: [], elements: [], previewBlocks: [], mapPreviewBlocks: Map(1), blockActive: null, baseContainer: {…} }
```

Ilustración 12: Resultado del middleware logger en la consola

7.2.2 Problemas y soluciones

Durante el desarrollo de esta fase de la aplicación nos dimos cuenta de que con el estado actual iba a ser complicado acceder a ciertas propiedades de los bloques si existía anidación. Por esta razón, decidimos añadir al estado un mapa con los elementos que se añaden a la vista previa y el contendor base. Como clave para insertar en el mapa el contenedor base y los bloques decidimos usar un `uuid` (identificador único universal o universally unique identifier) que nos permite tener de un mismo bloque varias instancias con sus configuraciones.

Al añadir el mapa se produjeron efectos inesperados en la aplicación. Cuando la aplicación se usaba se producían cambios de estado inexplicables. El problema se encontraba en la función de copia profunda de objetos que, como ya comentamos, usábamos un pequeño "truco" para conseguir tener una función de copia sencilla y eficiente, que consistía en pasar el objeto a JSON y de JSON a objeto. Esta función es muy útil pero no permite la copia de mapas, puesto que son objetos complejos. Esto producía la pérdida de los datos del mapa. Conseguimos corregirlo reemplazando nuestra función de copia por una de una librería externa que sí que permitía la copia profunda de mapas.

7.3 Componentes y funcionalidad

El desarrollo de este apartado es sin lugar a duda el más sencillo de los cuatro. Esto se debe a que los componentes solo deben mostrar el estado y la lógica de los contenedores es relativamente sencilla

exceptuando la funcionalidad `drag and drop`, tema que trataremos un poco más adelante, y el renderizado del JSON del cual hablaremos en el siguiente apartado.

7.3.1 Implementación

Durante esta fase del desarrollo no hemos tenido problemas reseñables ni hemos necesitado tomar decisiones destacables. El único cambio frente a lo planeado ha sido modularizar la pestaña de personalización separándola en dos componentes uno con las funciones de clonar, eliminar y cerrar pestaña y otro con propiamente la funcionalidad de personalización para poder reutilizarlo. El componente de personalización a su vez se divide en varios componentes dependiendo qué personalizan.

7.3.1.1 Drag and Drop

Lo más importante al implementar esta funcionalidad es que existen dos tipos de objetos arrastrables y dos operaciones. Tenemos la operación añadir bloque que solo puede ocurrir si se arrastra el componente "bloque visual", puesto que todavía no se encuentra en la previsualización, y la operación mover un bloque que ocurre con los bloques que se encuentran en la previsualización. Estos pueden moverse de posición, moverse dentro de otro bloque o extraerlo fuera de él.

Para poder implementar esta funcionalidad teniendo dos tipos de datos es necesario poder distinguir cuál es cuál. Por esta razón, en el evento `onDragStart`, que es el primer evento que ocurre en una operación `drag`, aparte de añadir los datos es necesario incluir el tipo de datos que se está añadiendo.

Dicho esto, necesitamos un total de cuatro eventos: dos eventos `onDragStart`, un evento `onDrop`, y un evento `onDragOver`. Lo primero que necesitamos es añadir un evento `onDragStart` al componente "bloque visual". Aquí comienza todo, lo primero que hay que hacer es añadir algún bloque al componente "previsualización". Este evento es muy sencillo, lo único que hace es transferir el tipo de bloque `visualBlock` y los datos, que son el identificador del bloque (añadido al leer los bloques disponibles la aplicación) y el tipo (elemento o modulo). El desencadenante de este evento es arrastrar un elemento HTML. Para que sea posible es necesario añadir el atributo `draggable` al elemento que queramos que se pueda arrastrar. Con esto habríamos acabado el componente "bloque visual".

Lo siguiente es añadir los eventos restantes en el componente "previsualización". De los tres eventos que nos quedan, dos son muy sencillos. En el primero, `onDragOver`, simplemente hay que prevenir el comportamiento por defecto y en `onDragStart`, que es muy parecido al anterior, aunque hay que cambiar el tipo de bloque a `previewBlock` y los datos, además de añadir que evite la propagación del evento. Esto es necesario debido a que, como es posible que ciertos módulos contengan otros bloques, puede pasar que los cambios se produzcan en el bloque superior y no en el que habíamos arrastrado.

Para finalizar, es necesario realizar la implementación del evento `onDrop` que, pese a ser el más complejo, no es muy difícil. Básicamente se recoge el tipo de dato y posteriormente en base a eso se llama a un creador de acciones u otro (añadir o mover).

7.4 Renderizado

Es la funcionalidad que más tiempo y recursos ha consumido del proyecto debido a la complejidad y limitaciones del lenguaje. Consiste en traducir un JSON en JSX con cada modificación del estado en el que se vean afectados los bloques existentes de la previsualización.

7.4.1 Implementación

La implementación está divida en dos componentes, primero el componente previsualización, del que ya hemos realizado algunas funciones, y un componente dedicado a la visualización en la aplicación del contenedor base. Además, son necesarias dos funciones que se encuentran en el fichero utilidades. Estas valen para convertir el código base64 a HTML y aplicar las customizaciones.

7.4.1.1 Componente previsualización

Gran parte de este componente ya lo habíamos realizado en la fase anterior por lo que solo queda mostrar el componente dedicado a la visualización del contenedor base, al que hemos llamado "Email," que se pasa como `children` al componente previsualización añadiéndole los eventos que definimos anteriormente y otra información relevante como los bloques de la previsualización o su personalización.

Esto en un principio parece sencillo, pero al decidirse en tiempo de ejecución el componente para visualizar, el contenido no puede usarse la sintaxis común:

```
<Email events={events} custom={ ...this.props.container.custom } previewBlocks={this.props.previewBlocks} />
```

Por este motivo, tuvimos que consultar la documentación y buscar como modificar un componente pasado como hijo (`props.children`):

```
<Preview>
  <Email />
</Preview>
```

Encontramos un método de React que permitía clonar elementos JSX y pasarle nuevos parámetros (`props`), el resultado final fue el siguiente (el código ha sido simplificado):

```
render () {
   const Container = React.cloneElement(
     this.props.children,
     {
       custom: { ...this.props.container.custom },
       previewBlocks: this.props.previewBlocks,
       events: this.events
     }
   )
```

```
    return (
      <main>
        {Container}
      </main>
    )
}
```

7.4.1.2 Componente email

El componente email se encarga de crear la estructura que seguirán los bloques. En nuestro caso es una tabla con un único sitio para establecer los bloques, por lo que deja total libertad al diseñar.

Básicamente lo que hace este componente es usar los datos y eventos que se le pasan por parámetro para crear el renderizado completo, usando las funciones del archivo utilidades para posteriormente mostrarlo.

7.4.1.3 Funciones auxiliares

Las dos funciones necesarias para el renderizado son: `JSONToHTML` y `base64ToHTML`.

La primera se encarga de recorrer los hijos del bloque descrito en JSON pasado por parámetro y llamar a `base64ToHTML` en cada uno de ellos. Posteriormente añade los hijos al bloque padre.

La función `base64ToHTML` se encarga de decodificar el código base64 y aplicarle las propiedades descritas en el JSON y posteriormente devuelve el HTML.

7.4.2 Problemas y soluciones

El código mostrado y la explicación de los componentes anteriores solo es válido para bloques que no permiten anidación. Estuvimos probando esta manera de trabajar con bloques que permitían anidación y se complicaba demasiado. El problema reside en que React no es capaz de interpretar el código HTML, una vez decodificado de base64, como por ejemplo si hace jQuery, si no, que lo entiende como un string complicando el trabajo absurdamente.

Un ejemplo de este problema lo encontramos al intentar insertar eventos en los hijos de un bloque, ya que era necesario insertar los eventos en el objeto `window` de JavaScript para que funcionasen.

Por estas razones decidimos pivotar la manera en que se representan los bloques y usar el JSON alternativo que analizamos en la etapa de diseño. Esta forma de describir los bloques es descriptiva y gracias a esto se puede crear dinámicamente los elementos JSX necesarios.

A pesar de que la nueva forma de representar los bloques nos ha facilitado el desarrollo sigue teniendo limitaciones impuestas por React: ciertas propiedades CSS no se renderizan y los comentarios y códigos HTML tampoco.

7.4.2.1 Redux y JSON

Al realizar un cambio en el método de renderizado puede provocar que tengamos que realizar cambios en Redux, aunque, debido a como está planteado el almacén, los cambios no son muy

grandes. Toda la estructura debería de servir con el nuevo método de renderizado, solo habría que cambiar la forma de tratar los datos en los creadores de acciones. Podríamos llegar al caso de tener que cambiar los parámetros de los creadores de acciones, pero no sería necesario realizar nada más.

Como ya habíamos dicho usaremos el JSON descriptivo que analizamos en el apartado de diseño, pero además queremos añadirle una opción para añadir comentarios HTML. Además, consideramos añadir un atributo que describa propiedades que estén exclusivamente en el diseñador para dar mayor flexibilidad visual pero finalmente no lo incluimos.

Además, como último cambio el JSON del contendor base contendrá cómo debe exportarse, descrito en la propiedad `exportContent` (guardado en base64), además de cómo debe verse en el diseñador. Este cambio es necesario para evitar las limitaciones de React y JavaScript.

7.4.2.2 Nuevo método de renderizado

El nuevo método de renderizado en un principio se componía de cinco funciones: `JSONParserToJSX`, `JSONApplyEvents`, `JSONAddHTMLCode`, `JSONAddHTMLComment` y `JSONApplyProperties`.

Esto era necesario porque queríamos renderizar los comentarios y los códigos HTML, pero decidimos prescindir de ellos debido a que provocaban errores en la aplicación y no eran características realmente útiles en el renderizado. Finalmente, eliminamos el método que aplicaba los eventos que se encontraban en el JSON. Sin lugar a duda, creemos que este paso era necesario para facilitar el desarrollo de nuevos bloques, puesto que ya no es necesario añadir los eventos (drag and drop) manualmente, si no que la aplicación se encarga de ello.

En resumidas cuentas, después de suprimir las funciones innecesarias el método de renderizado se compone de dos funciones: `JSONParserToJSX` y `JSONApplyProperties`.

La función principal es `JSONParserToJSX`, funciona recursivamente y usa `JSONApplyProperties` para aplicar las propiedades. La función tiene cinco parámetros de los cuales los tres últimos son optativos:

- `JSON`: a través de este parámetro se introduce el JSON que debe ser renderizado
- `JSX`: se usa para inicializar la variable JSX en un principio y posteriormente para transmitir el código JSX renderizado hasta que se devuelva
- `uuidLastBlock`: se almacena en él el uuid del último bloque. Por defecto tiene el valor `undefined`
- `eventProperties`: almacena los datos necesarios para aplicar los eventos correctamente. Por defecto es un objeto vacío
- `childrenArray`: sirve para mantener los hijos de los bloques. Es un array de arrays cuando se añade algún hijo. Por defecto es un array vacío

```
JSONToJSX = (json, jsx, lastUUID = undefined, evProp = {}, chArr = []) => {
  if (json.uuid) lastUUID = json.uuid
  if (!evProp.uuid) evProp.uuid = lastUUID
  json = JSONApplyProps(json)
  const props = {...json.elementConfig}
  const elValue = AllHtmlEntities.decode(json.elementValue)

  // Base case, JSON doesn't have content or is {this.props.children}
  if (!json.isArr() && (!json.content || json.content == '{children}')) {
    // To prevent crash when json only has comments
    if (json.elementType) {
      // No has children
      if (!json.content) {
        // HTML tag with values
        jsx = createElement(json.elementType, props, [elValue])
      } else {
        // Has children (content: {this.props.children})
        const children = chArr.shift()
        const childElements = [elValue]
        childElements.push(children.array)
        jsx = createElement(json.elementType, props, childElements)
      }
    }
    return jsx
  } else {
    // Has children
    if (json.children) {
      for (const [numGap, gap] of json.children.entries()) {
        const chOfGap = []
        for (const [nCh, children] of gap.entries()) {
          const evPropChild = {uuid: json.uuid, gap: numGap, children: nCh}
          chOfGap.push({JSONToJSX(children, jsx, lastUUID, evPropChild)})
        }
        chArr.push({gap: numGap, array: chOfGap})
      }
    }

    // Has content
    if (json.content && Array.isArray(json.content)) {
      const callback = []
      // Render brothers tags
      for (const [numNT, next] of json.content.entries()) {
        evProp = {uuid: next.uuid, gap: 0, children: numNT}
        callback.push({JSONToJSX(next, jsx, lastUUID, evProp, chArr)})
      }

      // Create element and add all nested tags
      if (json.elementType) {
        jsx = createElement(json.elementType, props, [elValue, callback])
      }
    }

    // JSON doesn't have elementType so it is a container or a block
    if (!json.elementType) {
      jsx = JSONToJSX(json.content, jsx, lastUUID, evProp, chArr)
    }

    return jsx
  }
}
```

Lo primero que realiza la función es comprobar si el JSON tiene un uuid, lo que significa que es el comienzo de un bloque. Si lo tiene lo asigna al parámetro `uuidLastBlock`. Posteriormente se aplican las propiedades.

A continuación, se comprueba si cumple el caso base, que básicamente es que sea el último objeto del JSON, en cuyo caso se crea el último nodo y se devuelve el JSX. En caso contrario, puede ocurrir que si tiene hijos se ejecute un bucle que lanza tantas llamadas a `JSONParserToJSX` como hijos tenga y añade las respuestas a un array. Cuando finaliza el bucle se añade el array que contiene todos hijos al parámetro `childrenArray`. Con esto se consigue que cuando entre en el caso base puedan añadirse sus hijos. Después de comprobar si tiene hijos, se comprueba si tiene contenido, de tenerlo se lanzan todos los objetos hermanos y se recogen. Posteriormente, si contiene la propiedad `elementType` el objeto JSON se crea un elemento JSX con ese tipo, configuración y por último los elementos anidados.

Para finalizar, ya fuera de la comprobación del caso base y el caso contrario, se comprueba si tiene tipo el bloque y en caso de no tenerlo se lanza `JSONParserToJSX`, puesto que se trata de un contenedor o bloque. Por último, se devuelve el JSX.

Todo este proceso se produce dentro de la función `JSONParserToJSX` , aunque se ha simplificado un poco para facilitar la explicación.

Para acabar, la función `JSONApplyProperties` aplica las propiedades que hay en el atributo `elementConfig` del JSON. Básicamente convierte el contenido en un string y posteriormente recorre las propiedades y las sustituye en el contenido por sus valores. Antes de devolver el JSON, convierte el contenido otra vez a objeto y lo asigna. Por último, devuelve el JSON.

7.5 Interfaz de usuario

En esta sección vamos a explicar todas las tecnologías que se han usado para realizar la interfaz web. Hemos dividido la sección en tres apartados. Primero hablaremos de los conceptos generales que se usan, posteriormente veremos ITCSS y acabaremos viendo cómo se han implementado los temas.

7.5.1 Conceptos generales

Los conceptos generales o tecnologías que son necesarios para entender el código CSS que hemos realizado son las siguientes:

- SASS: podría considerarse que es CSS con “superpoderes”. SASS nos permite anidar el código para una mayor claridad, uso de variables, uso de mixin (funciones) lo que nos permite extraer partes del código CSS que se repiten o son complicadas de hacer, uso de partials (snippets CSS que posteriormente pueden incluirse) entre otras opciones. El único punto negativo es que debe pre-procesarse el código para que se compile a CSS (lo realiza automáticamente `create-react-app`).

- Variables CSS: cuanto más complejo es el código CSS más necesarias son, puesto que facilitan la legibilidad del código y permiten cambiar propiedades CSS dinámicamente. Pueden usarse para guardar un color o el tamaño de una fuente, por ejemplo. Al contrario de las variables SASS que son sustituidas por sus valores en el pre-procesado, éstas no se ven afectadas al ser una funcionalidad nativa.
- icon-fonts: los iconos se usan mediante fuentes en vez de mediante imágenes. Esto nos permite editar su tamaño, color, modificar sus estilos y reducir el número de peticiones HTTP (van todos los iconos en un fichero). El único punto negativo es que solo pueden ser de un color, aunque FontAwesome ha dado un gran paso y permite que sus fuentes de pago puedan ser de dos colores.
- SMACSS: consideramos importante que todas las propiedades sigan el mismo orden, por ello decidimos seguir la guía de estilos SMACSS. SMACSS organiza las reglas CSS en 5 categorías: Box, Border, Background, Text y Other. Como ordenar los CSS es un proceso tedioso, decidimos automatizarlo por medio del paquete NPM `css-declaration-sorter`, en el cual colaboramos en su proyecto en GitHub aumentando el número de ejemplos y propiedades que soporta.

7.5.2 ITCSS

ITCSS (Inverted Triangle architecture for CSS) es una propuesta para la organización del código CSS que pretende facilitar el trabajo con las hojas de estilo en cascada. Es un modelo de arquitectura que pretende satisfacer de una manera estándar las necesidades comunes de los proyectos.

ITCSS son recomendaciones, por lo que se puede adaptar o modificar según las necesidades de la tecnología o el proyecto. En concreto hemos seguido la estructura de carpetas que puede verse en la siguiente imagen.

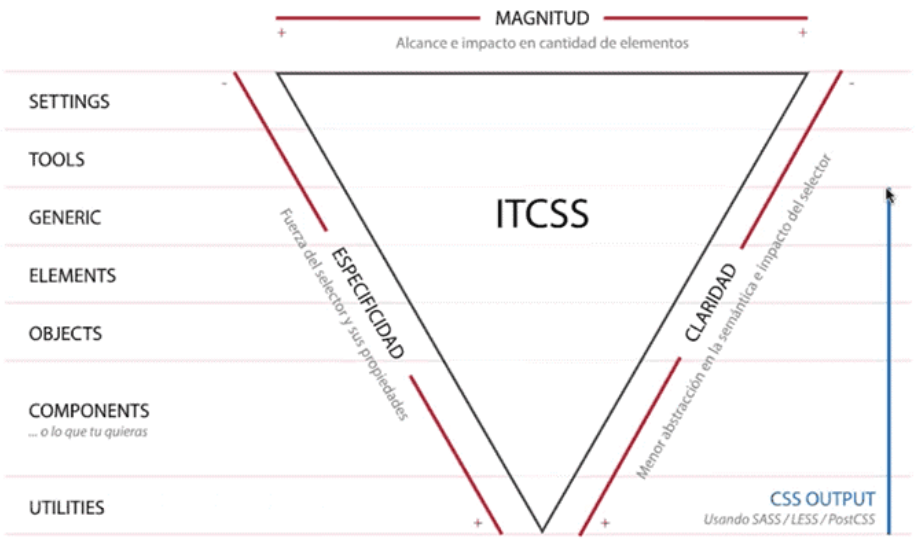

Ilustración 13: Representación gráfica de ITCSS

A continuación, vamos a explicar qué contiene cada carpeta más detalladamente.

- **Settings**: en esta carpeta colocaremos código de declaración de variables
- **Tools**: en esta carpeta tendremos los mixins y funciones, también de los preprocesadores
- **Generic**: en este punto colocaremos código genérico de base para todo el proyecto, típicamente el reset de CSS que estés usando, ej.: normalize
- **Elements**: nos servirá para colocar el CSS que afecta a los elementos del HTML. No vamos a colocar más que estilo para etiquetas, es decir, aquí no definimos estilo para clases (class) CSS
- **Object**: es donde irían las clases (class) para la estructura de la página. En esta carpeta es el lugar donde colocar aspectos relacionados con el layout, mediaqueries básicas para acomodar las partes de nuestra plantilla a distintas dimensiones de pantallas
- **Components**: aquí ya descendemos a estilos que van a afectar a diversos bloques concretos de nuestro documento, navegador, botones, recuadros de utilidades diversas, migas de pan, cabecera, buscador, botones sociales, etc. Esta es la carpeta que generalmente contendrá más archivos, pues cada pequeña parte de nuestra herramienta tendrá un archivo donde se defina su aspecto
- **Utilities**: por último, la carpeta donde colocaremos utilidades que sobrescriban estilos importantes, quizás no compatibles con todos los navegadores, en los que reformular aspectos como márgenes, alineamiento, posicionamientos con flexbox, font-face, etc. Este es el único sitio lícito en ITCSS para colocar la declaración CSS `!important`

Las carpetas object y components de nuestro proyecto se encuentran vacías, puesto que en React cada componente debe ser autocontenido. Esto, unido al uso de una distribución de carpetas orientado a la funcionalidad, nos llevó a preferir colocar los componentes fuera de la carpeta assets junto con el resto del código escrito en React.

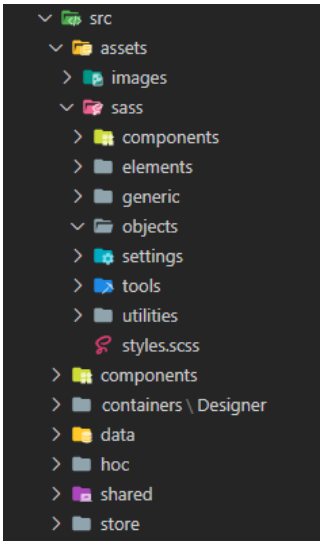

Ilustración 14: Estructura de carpetas del proyecto

7.5.3 Temas

La aplicación dispone actualmente de dos temas, uno claro y otro oscuro. Los temas se han definido con variables CSS, por lo que se pueden cambiar de manera sencilla en tiempo de ejecución agregando una clase que sobrescribe las variables. Esto también permite añadir fácilmente temas nuevos, puesto que para hacerlo solo es necesario crear un fichero definiendo las variables que deseemos.

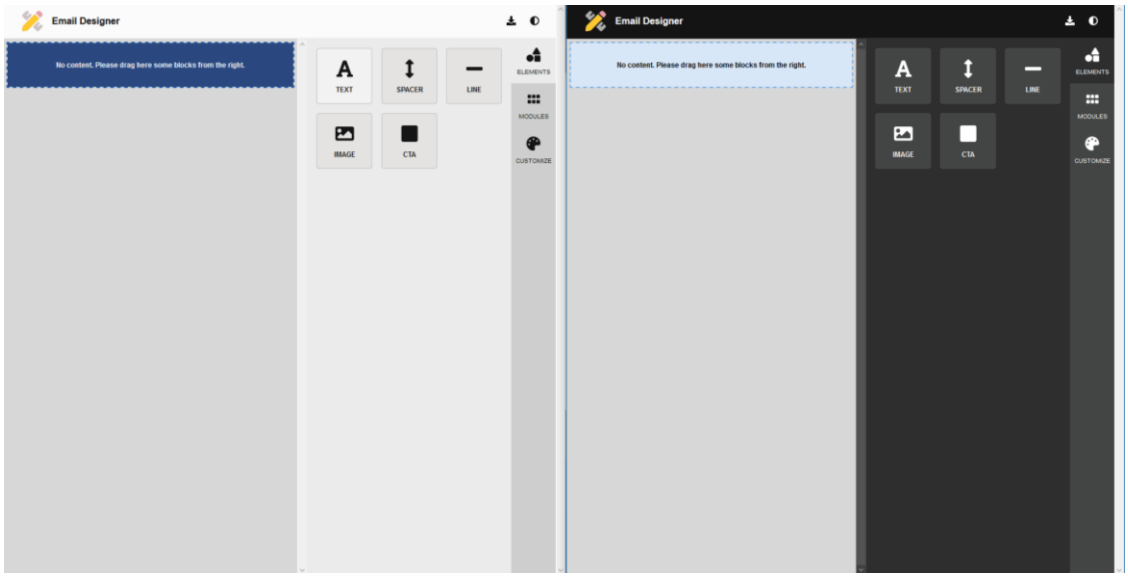

Ilustración 15: Temas disponibles en la aplicación

Para evitar que hubiese que elegir el tema cada vez que se iniciase la aplicación, decidimos guardar las preferencias en el almacén del navegador. Realizamos esto creando un nuevo almacén Redux donde se cargan las preferencias al iniciar la aplicación. A su vez, cuando se modifican las preferencias, se guardada tanto en el estado actual (que modifica la aplicación para que corresponda con el nuevo estado), como en el almacén del navegador.

8 Evaluación de la aplicación

Este apartado lo vamos a dedicar a la evaluación de la aplicación construida. Para ello disponemos de distintos mecanismos. Lo primero que trataremos son las pruebas unitarias, posteriormente realizaremos las pruebas de integración, después de estas tocará el turno de realizar las pruebas de funcionalidad y por último las pruebas de aceptación. Aunque en principio está previsto realizar todas estas pruebas, puede que alguna no sea posible realizarla y se tenga que descartar.

8.1 Pruebas unitarias

Las pruebas unitarias son de bajo nivel, cercanas al código fuente. Básicamente lo que hacen es probar funciones o métodos individuales de cada componente, clase o módulo utilizado por la aplicación. Por lo general, estas pruebas son sencillas de automatizar y de ejecutarlas con integración continua.

Es por ello por lo que decidimos integrar las pruebas en nuestro repositorio, que actualmente se encuentra en GitHub, gracias a los actions. La forma de realizarlo es muy sencilla. Para poder llevarlo a cabo es necesario crear un fichero en formato YAML, un formato de serialización de datos legible por humanos, en la ruta `.github/workflows/` donde describiremos cómo debe ser el action. En concreto nuestro action luce así:

```yaml
name: test
on:
  push:
    branches:
      - develop
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [8.x, 10.x, 12.x]
    steps:
    - uses: actions/checkout@v1
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - name: Install Packages
      run: npm install
    - name: Run Tests
      run: npm run test-all
```

El action es muy sencillo. Lo primero que vemos es el nombre y posteriormente se define cuándo debe lanzarse. En nuestro caso ocurre cuando se produce un push sobre la rama develop. Posteriormente en Jobs se define qué debe hacer. Como se ve, levanta la última imagen de Ubuntu disponible y además inicializa una matriz con las distintas versiones de node especificadas.

En el último apartado, steps, se especifican los pasos que debe hacer para llevar a cabo las pruebas. Primero se le indica que use una imagen que contiene node, posteriormente se instalan todos los paquetes node necesarios y finalmente ejecuta un script personalizado que lanza las pruebas. Fue necesario

realizar este script personalizado porque por defecto `npm test` necesita que se le introduzcan por consola cómo deben ejecutarse las pruebas.

Respecto a las pruebas realizadas, se ha comprobado el correcto funcionamiento de todos los creadores de acciones, que son los que controlan cómo evoluciona el estado de la aplicación. Hemos podido realizar estas pruebas gracias al flujo de trabajo que proporciona Redux, como ya hablamos anteriormente. Además, hemos usado la librería Jest, un marco de prueba de JavaScript mantenido por Facebook, para facilitar la creación de pruebas.

8.2 Pruebas de integración

Las pruebas de integración verifican que los distintos módulos o servicios utilizados por la aplicación funcionen correctamente juntos. Este tipo de pruebas son más costosas que las pruebas unitarias puesto que necesitan que varias partes de la aplicación estén activas.

Después de considerarlo hemos decidido no realizar ninguna prueba de integración puesto que la aplicación no se ha conectado al final con ninguna API para obtener los bloques y no existe un control de acceso o alguna otra funcionalidad independiente que se integre con la funcionalidad principal.

8.3 Pruebas de funcionalidad

Las pruebas de funcionalidad se encargan de comprobar que se cumplen los requisitos de una aplicación. Por ejemplo, en nuestro caso, podríamos verificar que puede exportarse en HTML un correo creado. La prueba no se preocupa de cómo se exporta, si no del resultado final.

En concreto hemos realizado las siguientes pruebas:

- Añadir bloques al contenedor base
- Personalizar, duplicar y eliminar los bloques
- Personalizar el contenedor base
- Cambiar el tema y que se inicialice automáticamente la aplicación con el tema seleccionado al recargar la web
- Descargar el correo electrónico en formato HTML

8.4 Pruebas de aceptación

Las pruebas de aceptación son pruebas formales que se usan para verificar si la aplicación cumple los requisitos del cliente.

Al haber renunciado el cliente inicial y ser nosotros mismos junto con el tutor el cliente, no consideramos necesario realizar pruebas de aceptación.

9 Seguimiento y control

A pesar de la realización de una planificación, el desarrollo del proyecto ha variado de lo planificado en varias ocasiones. A lo largo de este apartado analizaremos cuáles han sido los cambios más importantes que han afectado al proyecto, cómo han afectado al desarrollo y sus consecuencias directas e indirectas.

Al comienzo del proyecto planificamos su duración y fijamos el total de horas de trabajo en trescientas. Además, incluimos cinco hitos que nos servirán como puntos de control para ver cómo ha ido evolucionando el proyecto.

Inicialmente no disponíamos de etapas, sino de hitos como hemos mencionado. Pero para facilitar la explicación de cómo se ha consumido el tiempo, hemos decidido crear etapas que se correspondan con el diagrama de Gantt. En la siguiente tabla se encuentran las distintas etapas que hemos definido:

Etapa	Inicio	Finalización	Horas
1. Viabilidad del proyecto	03/02/2020	17/02/2020	41.5h
2. Primera versión funcional	17/02/2020	06/04/2020	110.5h
3. Segunda versión funcional	06/04/2020	05/05/2020	79h
4. Tercera versión funcional	05/05/2020	19/05/2020	30h
5. Finalización del proyecto	19/05/2020	12/06/2020	39h

Continuando con el diagrama de Gantt de la planificación, las horas dedicadas se dividen en varias fases del proyecto:

- Planificación: 20h
- Análisis: 13h
- Diseño: 40h
- Implementación: 117h
- Pruebas: 35h
- Seguimiento y control: 15h
- Conclusiones: 10h
- Documentación: 50h

El trabajo que se realiza en cada etapa es el siguiente:

- Etapa 1: planificación y análisis
- Etapa 2: diseño e implementación
- Etapa 3: implementación
- Etapa 4: implementación
- Etapa 5: conclusiones, presentación y lecciones aprendidas

Además, durante todas las etapas se llevará a cabo la redacción de la memoria y el proceso de seguimiento y control del proyecto.

Una vez dejado claro cómo hemos divido el proyecto y qué tareas se realizaron en cada fase, vamos a exponer los principales contratiempos que han producido las mayores desviaciones. Además, analizaremos sus consecuencias y contabilizaremos su coste.

Durante la primera etapa el proyecto avanzó más rápido de lo esperado. El cliente sabía qué quería y tenía una idea clara de qué necesitaba y cómo quería hacerlo. Esto se tradujo en un pequeño colchón ante imprevistos, que aumentó al ya existente de diez días (al planificar acabar el 12/06/2020 y ser la entrega del 22 al 24/06/2020).

La siguiente desviación se produjo durante la etapa dos. Mientras implementábamos el sistema de renderizado de la aplicación, surgió el problema de que React no genera un "envoltorio" que facilite trabajar con código HTML almacenado en una variable de tipo string, como por ejemplo si hace jQuery. Esto provocó que al almacenar el código de los bloques en base64 fuese inviable continuar debido a su complejidad. Para solucionar este problema decimos rehacer la función de renderizado y los bloques para que funcionasen usando el JSON descriptivo. En consecuencia, parte del tiempo que teníamos reservado para imprevistos fue consumido por este imprevisto. Más adelante presentaremos una tabla con datos más concretos.

Por último, la desviación más grande del proyecto se produjo también durante esta etapa y se mantuvo en el resto. Esta se produjo por una causa imprevista como ha sido la pandemia del coronavirus Covid-19. Una de las consecuencias de esta pandemia fue la interrupción del proyecto en la empresa Pixel Division. Esto nos puso en la tesitura de continuar el proyecto sin el cliente o empezar uno nuevo. Aunque el plan de contingencia no contemplaba una pandemia adaptamos el plan de enfermedad a esta situación, produciendo una replanificación y la implantación de teletrabajo. Durante aproximadamente un mes estuvimos trabajando de esta manera a la espera de alguna comunicación de Pixel Division. A finales de marzo tomamos la decisión de continuar con el proyecto convirtiéndose en mi cliente el tutor académico del proyecto, debido al avanzado estado del proyecto. Después de comunicárselo a la empresa, a mediados de abril fue "oficial" la propuesta.

Además, del cambio de cliente hubo otras consecuencias. Con este cambio las pruebas de aceptación eran prescindibles, lo que nos dio un respiro debido a los retrasos. También se produjo una reducción del control de horas debido a la situación y la pérdida de la remuneración económica. Otro impacto en proyecto fue la imposibilidad de completar los bloques previstos al no tener el permiso de uso de los bloques de la empresa.

Para mayor claridad, hemos realizado un gráfico que puede verse en la ilustración 17. En él se refleja el porcentaje del trabajo realizado en cada etapa (eje izquierdo) y el tiempo invertido en cada etapa (eje derecho).

Queremos destacar que cuando se planificó el proyecto se puso la mayor carga de trabajo al inicio y disminuye en las etapas finales. Esta decisión contribuyó a que, pese haber desviaciones importantes, se haya podido acabar a tiempo el proyecto. Puede verse más claro en la ilustración 16.

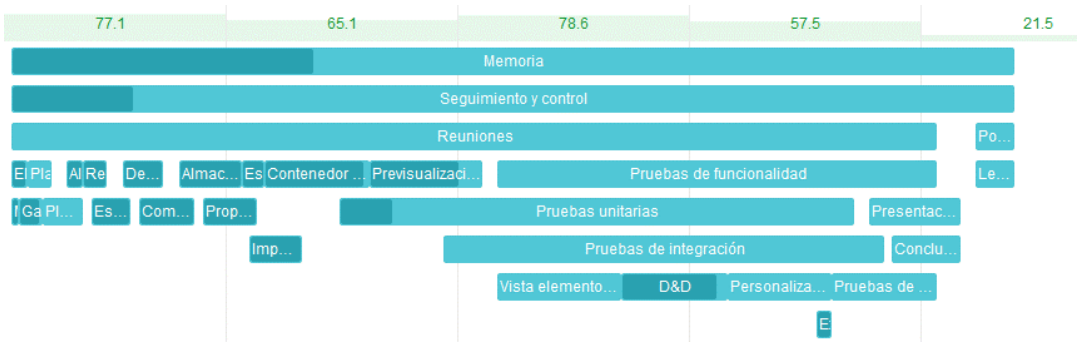

Ilustración 16: Carga de trabajo/mes

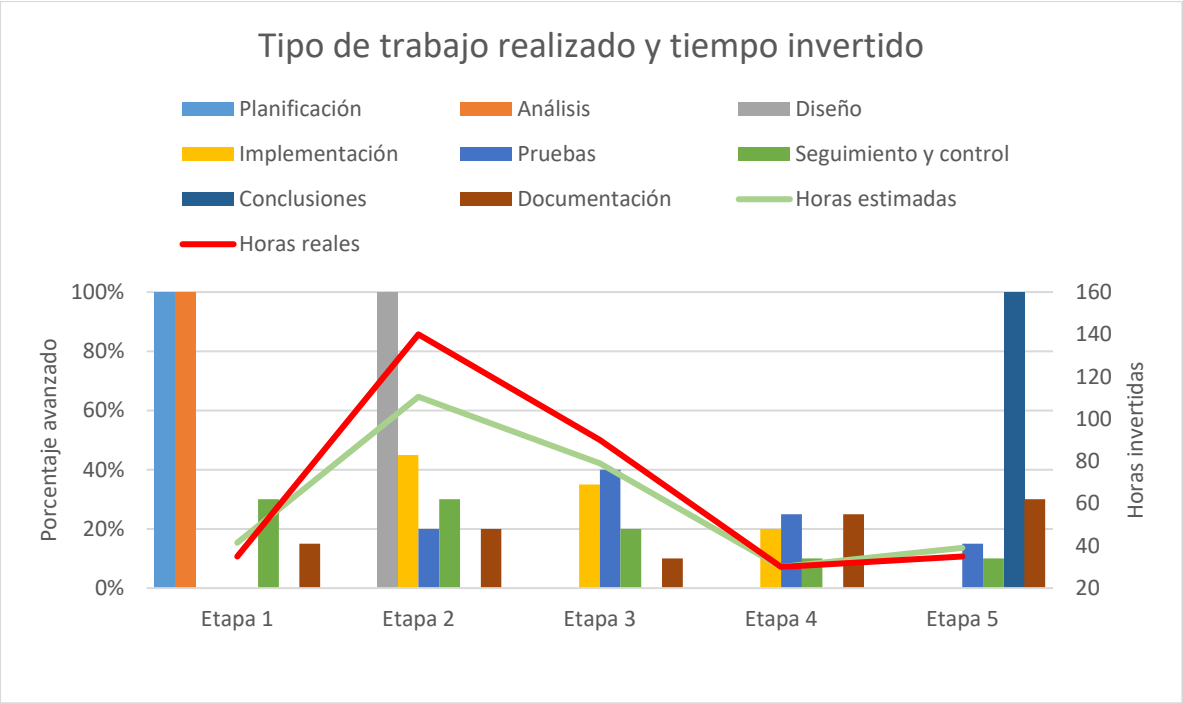

Ilustración 17: Gráfica que relaciona el trabajo realizado y las horas invertidas

Antes de realizar la conclusión, queremos mostrar las desviaciones que se han producido en cada fase a lo largo del proyecto, pueden verse a continuación en la siguiente tabla.

Etapa	Horas estimadas	Horas reales	Desvío
Etapa 1	41,50	35,00	-15.66%
Etapa 2	110,50	140,00	26,70%
Etapa 3	79,00	90,00	13.92%
Etapa 4	30,00	30,00	0,00%
Etapa 5	39,00	35,00	-10,26%

En conclusión, a pesar de las dificultades que han surgido durante el proyecto, ha sido posible llevarlo a buen puerto con un sobrecoste no muy superior al estimado, siendo este de un 10%, por lo que consideramos que ha sido un éxito el proyecto.

10 Lecciones aprendidas

Las lecciones aprendidas pueden definirse como el conocimiento adquirido en base a las experiencias vividas durante el desarrollo del proyecto. Esas experiencias junto con la reflexión y el análisis de lo sucedido generan las lecciones aprendidas, donde se expone el problema encontrado y cómo se ha solucionado. De esta manera el equipo es consciente de los errores y aciertos en futuros proyectos.

10.1 Ordenar automáticamente las propiedades CSS

Mantener un código CSS de calidad es difícil. Sin darte cuenta acabas con cientos de ficheros desordenados, propiedades que se sobrescriben o desorden generalizado. En definitiva, un caos.

Existen múltiples técnicas para reducir este riesgo, pero en concreto en esta lección vamos a unificar la ordenación de las propiedades CSS, lo que facilitará encontrarlas cuando sea preciso y no repetir propiedades.

10.1.1 Prerrequisitos

Los prerrequisitos para instalar el software necesario para ordenar automáticamente las propiedades son mínimos. Solo es necesario tener instalado Node 10 o superior y NPM.

10.1.2 Instalación

Una vez tengamos instalado Node, NPM e inicializado un proyecto NPM (si no usar `npm init`), solo debemos instalar los siguientes paquetes: `postcss-cli`, `postcss-scss` (solo es necesario si se usa SASS) y `css-declaration-sorter`. En concreto los instalaremos como dependencias de desarrollo puesto que no son necesarios para distribución. Esto puede realizarse de la siguiente manera en una terminal:

```
npm install postcss-cli postcss-scss css-declaration-sorter –save-dev
```

Posteriormente tendremos que definir la configuración de PostCSS para que use el plugin `css-declaration-sorter`, que es quien realiza la "magia". Esto puede hacerse en el fichero `package.json` añadiendo:

```
"postcss": {
  "syntax": "postcss-scss",
  "map": false,
  "plugins": {
    "css-declaration-sorter": {
      "order": "smacss"
    }
  }
}
```

Con esta configuración se usará la sintaxis SCSS de SASS y se ordenará siguiendo SMACSS. Además, el plugin tiene más formas de ordenar las propiedades, incluso pudiendo crear la tuya propia. Para más información podemos consultar el repositorio.

Una vez realizados todos estos pasos ya podemos ordenar automáticamente nuestros CSS con el siguiente comando:

```
postcss ./src/**/*.scss --replace --config package.json
```

También puede crearse un script en el `package.json` o incorporar el anterior comando a la ejecución del script `npm start`.

10.2 Simplifica el uso de BEM usando mixins

El código CSS según va creciendo se complica. Para remediar esto surgieron distintas técnicas, entre ellas BEM (Block Element Modifier), una metodología que ayuda a crear componentes reutilizables y compartir código en el desarrollo front. Usar BEM es muy sencillo, solo es necesario aplicar la nomenclatura BEM, pero se hace tedioso puesto que cuanto más profundizas en el árbol de elementos más largos son los nombres de las clases. Una solución sencilla para remediar esto es usar los siguientes mixins:

```
/// Block Element
/// @access public
/// @param {String} $element - Element's name
@mixin element($element) {
  &__#{$element} {
      @content;
  }
}

/// @alias element
@mixin e($element) {
  @include element($element)  {
      @content;
  }
}

/// Block Modifier
/// @access public
/// @param {String} $modifier - Modifier's name
@mixin modifier($modifier) {
  &--#{$modifier} {
      @content;
  }
}

/// @alias modifier
@mixin m($modifier) {
  @include modifier($modifier) {
      @content;
  }
}
```

Gracias a los mixins se simplifica enormemente el uso de BEM. Puede verse un ejemplo a continuación:

```
.block {
  .block__element {
    .block__element--modifier {
      color: red;
    }
  }
}
```

```
.block {
  @include e(element) {
    @include m(modifier) {
      color: red;
    }
  }
}
```

10.3 Busca distintos enfoques para resolver un problema

Durante un proyecto pueden surgir retos y problemas, pero lo importante es no hundirse en un vaso de agua y saber encontrar soluciones. La mayoría de las veces solo es necesario saber buscar correctamente en internet. Y si por algún casual no encuentras la solución, lo más probables es que si acudes a la documentación casi seguro la hallarás. Si a pesar de todo sigues sin encontrarla, replantea el problema y piensa en alternativas.

A pesar de ser obvia esta lección aprendida la consideramos importante porque parece que no la aprendes hasta que te sucede.

10.4 Automatización de la integración continua con GitHub Actions

GitHub Actions facilita la automatización de todos los flujos de trabajo. Con esta tecnología podemos construir, pasar test e incluso desplegar las soluciones software desarrolladas. En concreto, nosotros lo hemos usado para pasar las pruebas cada vez que se hacía un `push` sobre la rama `develop` y publicar la última versión de la herramienta en GitHub Page.

Para conseguir esto solo es necesario crear el directorio `.github/workflow` en la raíz del proyecto y añadir los Actions que necesitemos. En concreto nosotros hemos usado dos.

El primero se encarga de pasar las pruebas sobre `develop` y el segundo de pasar las pruebas y desplegar la herramienta cuando se produce un `push` a `master`. El código que hemos usado para la rama `master` es el siguiente:

```
name: deploy
on:
  push:
    branches:
      - master
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [8.x, 10.x, 12.x]
    steps:
    - uses: actions/checkout@v1
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - name: Install Packages
      run: npm install
    - name: Run Tests
      run: npm run test-all
    - name: Deploy to GH Pages
      run: |
        git config --global user.email "${{ secrets.GIHUB_EMAIL }}"
        git config --global user.name "${{ secrets.GITHUB_NAME }}"
        git remote set-url origin ${{ secrets.GITHUB_URL }}
        npm run deploy
        echo "Deployed successfully"
```

11 Conclusiones

La generación automática de código es ampliamente conocida, sobre todo en el mundo web. Grandes empresas como Adobe, 1&1 y Google, tienen servicios de este tipo. Es cierto que la gran mayoría de estas herramientas ofrecen resultados vistosos, pero lo hacen a través de generar un código complicado, difícil de mantener, mal estructurado o incluso, a veces, sin sentido. En este contexto emprendimos el proyecto, siendo conscientes de que la mejor forma de realizarlo, en nuestra opinión, era independizar la herramienta de generación de código de los bloques que usa y de su contenedor base. De esta manera, logramos que la calidad del código generado no dependa de la herramienta, sino de los bloques y del contendor base.

A lo largo del proyecto nos ha atormentado un problema recurrente: mostrar comentarios y códigos de caracteres HTML en React. Hemos empleado muchas horas buscando una solución y no la conseguimos hasta que comprendimos que no tiene por qué ser igual lo que se ve a lo que se exporta. Con este nuevo enfoque conseguimos solucionarlo rápidamente mostrando el carácter " " y exportando el código ` `.

Pese haber conseguido acabar el proyecto satisfactoriamente nos queda un sabor agridulce. Los correos electrónicos que se pueden realizar con la herramienta son poco atractivos, y no por culpa de esta, ya que, como hemos comentado anteriormente, eso depende de los módulos. Al haber dejado de colaborar con Pixel Division no disponemos de bloques vistosos.

Para concluir esta sección quisiéramos exponer lo que ha supuesto el proyecto para nuestro *know-how*. Lo primero que queremos destacar es que no solo hemos aprendido a usar React, si no que ha servido para mucho más. Nos ha enseñado la importancia del inglés que, pese a estar la documentación oficial en español, la mayoría de las librerías tienen su documentación en inglés y el curso de formación que tomamos en la empresa de más de 40 horas, fue íntegramente en inglés. Por esta razón, todo el desarrollo y la documentación se han elaborado en inglés para posteriormente publicarlo y que cualquiera pueda usarlo. Además, nos ha enseñado que desesperarnos por un problema no ayuda nada a solucionarlo y a veces es necesario otro enfoque para encontrar la solución. Pero no todo lo aprendido viene de una "mala" experiencia y es que la parte que más nos ha gustado, gracias a las prácticas en Pixel Division, ha sido todo lo que hemos aprendido sobre front y sus tecnologías que posteriormente hemos puesto en práctica desarrollando la interfaz gráfica del proyecto.

12 Bibliografía

Angel Alvarez, M. (17 de 06 de 2017). *ITCSS*. Recuperado el 15 de 04 de 2020, de desarrolloweb: https://desarrolloweb.com/articulos/itcss-que-es-bases.html

Fernández, G. (24 de 08 de 2018). *React: 4 tipos de componentes para gobernarlos a todos*. Recuperado el 15 de 02 de 2020, de Medium: https://medium.com/@ger86/react-4-tipos-de-componentes-para-gobernarlos-a-todos-7c8f5c28e0b0

Joy Lally, A. (02 de 08 de 2016). *An Obsession with Design Patterns: Redux*. Recuperado el 17 de 04 de 2020, de Zalando: https://jobs.zalando.com/en/tech/blog/design-patterns-redux/

Murphy, M. (08 de 08 de 2016). *Journey to Enjoyable, Maintainable Styling with React, ITCSS, and CSS-in-JS*. Recuperado el 15 de 04 de 2020, de Medium: https://medium.com/maintainable-react-apps/journey-to-enjoyable-maintainable-styling-with-react-itcss-and-css-in-js-632cfa9c70d6

Murphy, M. (08 de 06 de 2016). *React Apps: Approaching Organization / Structure / Architecture*. Recuperado el 15 de 04 de 2020, de Medium: https://medium.com/maintainable-react-apps/react-apps-approaching-organization-structure-architecture-49a281bd97eb

Pittet, S. (s.f.). *The different types of software testing*. Recuperado el 28 de 05 de 2020, de Atlassian: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

Ramón Rodríguez (coordinador), J., García Mínguez, J., & Lamarca Orozco, I. (2007). *Gestión de proyectos informáticos: métodos, herramientas y casos.* 08018 Barcelona: Editorial UOC.

React. (s.f.). *Componentes y propiedades*. Recuperado el 08 de 02 de 2020, de React: https://es.reactjs.org/docs/components-and-props.html

React. (s.f.). *Presentando JSX*. Recuperado el 20 de 02 de 2020, de React: https://es.reactjs.org/docs/introducing-jsx.html

Redux. (s.f.). *createStore(reducer, [initialState], [enhancer])*. Recuperado el 17 de 04 de 2020, de Redux: https://es.redux.js.org/docs/api/create-store.html

Redux. (s.f.). *Store*. Recuperado el 17 de 04 de 2020, de Redux: https://es.redux.js.org/docs/api/Store.html

Unlayer. (s.f.). *React emaileditor demo*. Recuperado el 10 de 04 de 2020, de netlify: https://react-email-editor-demo.netlify.com/

Weck, S. (12 de 03 de 2017). *Developing modern offline apps with ReactJS, Redux and Electron – Part 3 – ReactJS + Redux*. Recuperado el 17 de 04 de 2020, de Codecentric: https://blog.codecentric.de/en/2017/12/developing-modern-offline-apps-reactjs-redux-electron-part-3-reactjs-redux-basics/

Wikipedia. (s.f.). *Desarrollo en cascada*. Recuperado el 03 de 02 de 2020, de Wikipedia: https://es.wikipedia.org/wiki/Desarrollo_en_cascada